How to find his way in the jungle of consistency criteria for distributed objects memories (or how to escape from Minos'labyrinth)

Michel Raynal, Masaaki Mizuno

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

How to Find his Way in the Jungle of Consistency Criteria for Distributed Objects Memories (or how to Escape from Minos' Labyrinth)

Michel RAYNAL
Masaaki MIZUNO

N° 1962
Juillet 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

How to Find his Way in the Jungle of Consistency Criteria for Distributed Objects Memories (or how to Escape from Minos' Labyrinth)

Michel Raynal*, Masaaki Mizuno**

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet ADP

Rapport de recherche n° 1962 — Mai 1993 — 7 pages

Abstract: This paper surveys consistency criteria that have been proposed, and sometimes implemented, for shared distributed objects and memories. Linearizability, sequential consistency, hybrid consistency and causal consistency are particularly emphasized; some protocols implementing these criteria are described. It is suggested that the hybrid consistency frame, introduced by Attiya and Friedman, constitutes the Ariadne's clew to understand this jungle of consistency criteria.

(Résumé : tsvp)

This work was supported in part by the Commision of European Communities under ESPRIT Basic Research project number 6360 (BROADCAST)

This work was supported in part by the Natioanl Science Foundation under Grant CCR-9201645

*Michel.Raynal@irisa.fr
**Kansas State University, Manhattan, KS 66506, USA masaaki@cis.ksu.edu

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

À propos des critères de cohérence pour mémoires virtuelles réparties

Résumé : Cet article étudie, de façon critique, des critères de cohérences proposés pour les mémoires virtuelles réparties. La linéarisabilité, la cohérence séquentielle, la cohérence hybride et la cohérence causale sont particulièrement analysées. Il est suggéré que le modèle proposé par Attiya et Friedman, fondé sur la cohérence hybride, peut constituer l'élément clé pour une meilleure appréhension de ce problème de cohérence.

How to find his way in the jungle of consistency criteria for distributed objects memories (or how to escape from Minos' labyrinth)

Michel RAYNAL*
IRISA - Campus de Beaulieu
35042 Rennes Cédex - FRANCE
raynal@irisa.fr, fax: (33) 99.38.38.32

Masaaki MIZUNO†
Kansas State University
Manhattan, KS 66506, USA
masaaki@cis.ksu.edu

Abstract

This paper surveys consistency criteria that have been proposed, and sometimes implemented, for shared distributed objects and memories. Linearizability, sequential consistency, hybrid consistency and causal consistency are particularly emphasized; some protocols implementing these criteria are described. It is suggested that the hybrid consistency, introduced by Attiya and Friedman, constitutes the Ariadne's clew to understand this jungle of consistency criteria.

1 Introduction

A clear view on future trends of distributed computing systems cannot be developed only by proposing new ideas, new concepts, new methods, and new technologies. These are necessary but in any case not sufficient. To attain this goal, we also have to clarify our ideas about previous proposals (sometimes contradictory !) in distributed computing systems. In this paper we focus on one of these topics and give an Ariadne's clew to understand it and consequently to benefit from it when we are to design and build a distributed system which is based on it.

In this paper, we study consistency criteria that have been proposed to express and implement distributed shared memories. The paper consists of three main sections. Section 2 presents a distributed system model we consider: the system consists of multiple nodes interconnected by a communication network and provides shared object abstraction to application programs. The following two sections present consistency criteria that protocols managing shared objects have to implement. Section 3 addresses correctness criteria and shows that they lead to two classes of distributed memories based on the degree of the concurrency we want to allow: One class is characterized by the *linearizability* property (a well-known protocol by Li and Hudak is in this class). Another is characterized by a less stringent property called *sequential consistency*. Each class of distributed memory systems is precisely analyzed. Protocols that implements them are also presented. Section 4 addresses *hybrid consistency* that supports strong and weak operations. This framework was introduced by Attiya and Friedman and can not only help to understand several previous proposals, but also allow a system designer to "tailor" a distributed shared memory semantics well-suited to his requirement. Clear clarification of distributed shared memory models and semantics is helpful if one wants to take up the challenge of future distributed computing systems; this paper is a step towards such a clarification.

*This work was supported in part by the Commission of European Communities under Esprit Basic Research project Number 6360 (BROADCAST).

†This work was supported in part by the National Science Foundation under Grant CCR-9201645.

2 Distributed model

2.1 Application programs

We assume that Application programs are made up of a set of n processes $P_1, \ldots, P_n$. Processes run concurrently and communicate with one another via the shared memory.

The shared memory maintains a collection of objects. Each object can be accessed by some predefined operations. For our purpose, we consider that each of these operations can be reduced to either a read or a write operation. A read operation by process P_i on object x which returns value v is denoted $r_i(x)v$; similarly a write operation on y with value v by P_i is denoted $w_i(y)v$.

2.2 Distributed execution

During the execution of a distributed program, each process issues a sequence of read and write operations on a set of shared objects (objects local to each process are not considered). Such a sequence is traditionally called a *process history*. A distributed execution can be represented by a partial order on the set of all operation invocations. At the application level, reads and writes are indivisible. Several consistency criteria on shared memory systems are defined based on particular partial order sets to be allowed by the system and processes' views on the partial order sets.

2.3 Underlying hardware support

The underlying support that executes programs consists of a collection of nodes interconnected by a communication network. The node communicate with each other by exchanging messages through FIFO channels. There is neither central memory nor a global clock. The underlying system is reliable and asynchronous; that is, nodes execute processes at their own speed and message transfer delay is finite but unpredictable. Each node is endowed with a local memory. As we are not interested in load balancing or migration, in order to facilitate the presentation, we assume that there are n nodes each of which executes exactly one process.

2.4 Problem description

We are interested in designing a software layer which implements a shared memory abstraction. The software layer uses local memories of nodes and the communication network. It is defined by a protocol executed by each node. The protocol interprets all the read and write operations issued by a process running on the node. In order to implement a particular consistency criterion defined on the set of shared objects, the protocol also communicates with ones on other nodes.

3 Linearizability and sequential consistency

3.1 Basic idea

A centralized memory system (multiprocessor with a central memory) implements a classical *atomic consistency* criterion (also called *linearizability*): each object has a unique copy, all the write operations are totally ordered, and a read operation on an object x always returns the last value written into x. Furthermore, all the non-overlapping operations are performed in the order issued. With the advent of distributed memory parallel machines, some objects management protocols have been defined; they allow several copies of each objects to exist concurrently (in order to improve reads efficiency).

As a less restrictive form of consistency, sequential consistency was introduced by Lamport in [9]. The basic idea of the *sequential consistency* criterion is to provide each execution with a total order on all the invoked operations [5,7,9]:

- that is an interleaving of all the processes histories
- and that could have occurred in a real execution (this means the semantics of objects is not violated by the total order; that is sometimes called legality[1]).

The idea associated with sequential consistency is that there is a correct sequential execution of the distributed program that produces the same result as the distributed execution under consideration.

In this section, adopting the terminology of [7] and borrowing elements from [5], we formally define two classes of consistency criteria: linearizability and sequential consistency. We also present protocols that implement them.

3.2 Linearizability

3.2.1 Consistency criterion

In order to formally define linearizability, the following notation is used to denote real-time order in execution. A distributed execution σ is associated with a partial order $\xrightarrow{\sigma}$ on operations in execution defined by: $op_1 \xrightarrow{\sigma} op_2$ if operation op_1 is terminated (with respect to real-time) before op_2 begins.

An execution σ is said to be linearizable if the associated partial $\xrightarrow{\sigma}$ order can be extended to a total order τ and if the execution associated with τ is legal. In other words there is a legal sequential execution τ (1) that produces the same behavior as σ for all the processes and for all the objects, and (2) that preserves real-time order of operations (i.e. all operations that are ordered in σ are ordered in the same way in τ). This definition introduced by Herlihy and Wing [7] states formally what is sometimes called atomicity by several authors. Figure 1 displays an execution $\sigma 1$; let $\sigma 2$ be an execution similar to $\sigma 1$ except for the last operation of P_2 that is replaced by $r_2(x)b$. Executions $\sigma 1$ and $\sigma 2$ involve 3 processes and one object x; segments indicate real-time durations of operations (with real-time progressing from left to right). Execution $\sigma 1$

[1] more formally a sequence (or total order) of operations τ is legal if for every object x the restriction of τ to operations of x belongs to the serial specification of x [5,7]

is linearizable, while $\sigma 2$ is not. The legal sequence $\tau 1$ associated with $\sigma 1$ is:

$$\tau 1 = w_1(x)a\ w_2(x)b\ r_1(x)b\ w_3(x)c\ r_2(x)c$$

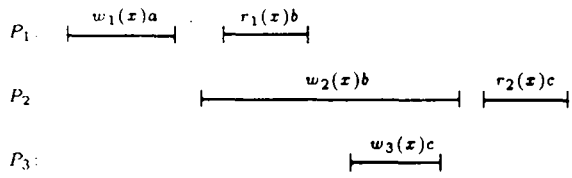

Figure 1: A linearizable execution $\sigma 1$

If $w_3(x)c$ overlapped $r_1(x)b$ then $\sigma 2$ would be linearizable as we would have:

$$\tau 2 = w_1(w)a\ w_3(x)c\ w_2(x)b\ r_1(x)b\ r_2(x)b$$

3.2.2 Protocols

Many of the protocols that implement distributed objects provide processes with local copies of objects and ensure the linearizability property. They use invalidation (write-once) or update (write-through) of copies to do that. To improve read efficiency an object can have several copies in nodes local memories. In the *write-through* method each write operation updates all the copies; these updates are done atomically. In the *write-once* method when an object is written all of its copies are atomically invalidated except for the one in the local memory of the writing process. When a process reads an object it first obtains a copy, if it does not have a copy in its local memory. One of the most known write-once protocols is proposed by Li and Hudak's in [10]. The protocol borrows its principle from the Berkeley cache consistency protocol, where an object is a page. Mechanisms based on notions of object owner (the last process that wrote it) and object manager (a process that knowns the last owner of the object and can thus indicate to others where to obtain a current copy) are used by the protocol. The reader is referred to [10] that gives a description and an evaluation of this protocol and to [3] that analyzes caches consistency protocols that ensure linearizability; finally [6] describes an implementation of the write-through method.

3.2.3 An important practical property

Herlihy and Wing proved a very important result that follows [7]: (using their terminology) *linearizability is a local property*. It means that a shared objects memory that provides the linearizability property can be implemented by a collection of separate protocols, each of which supports linearizability on a separate object (or set of objects) independently ([5] calls this property compositionality). This property is very important from a practical point of view: it states that linearizability copes naturally with the scaling problem when the number of objects increases.

3.3 Sequential consistency

3.3.1 Consistency criterion

An execution σ is sequentially consistent if there exists a legal sequence τ on all operations of σ such that for each process P_i, the restriction of τ to operations of P_i is the history of P_i. In others words, "sequential consistency" is "linearizability" minus "real-time order on non overlapping operations."

From a practical point of view, if an execution is sequentially consistent, all processes and all objects agree on a legal interleaving of operations but this sequence can differ from what really occurred; what is required is only that the commonly agreed sequence τ could have occurred. Consider Figure 2; $\sigma 3$ is not linearizable but is sequentially consistent since every process can agree on:

$$\tau 3 = w_3(x)a\ r_3(x)a\ w_2(y)c\ w_2(x)b\ r_1(x)b$$

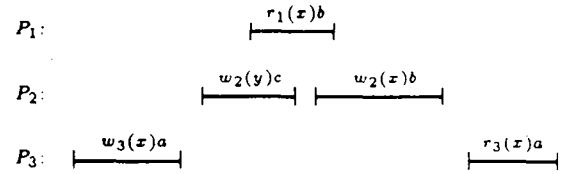

Figure 2: A sequentially consistent execution $\sigma 3$

In the concurrency control field the concept of *serializability* is used to express the same basic idea. If all transactions consisted of only either one read or one write on only one object then serializability and sequential consistency merge.

3.3.2 Protocols implementing sequential consistency

Several protocols implementing sequential consistency as correctness criterion for a set of objects have been proposed. We briefly present two of such protocols. Both protocols assume that each local memory contains copies of all the objects (this assumption is not mandatory, it only provides simpler protocols). The

first protocol is presented by Attiya and Welch [5]: it is based on an atomic broadcast facility at the system level. The second protocol, by Mizuno et al. [11], is based on a central object manager.

3.3.3 Protocol based on atomic broadcast

An atomic broadcast facility is a primitive (called *ab*) that guarantees the following delivery properties:

- every message sent is delivered at every process (including its sender)
- all messages sent are delivered in the same order
- two messages sent by the same process are delivered in their sending order

In the protocol description, we assume that only the delivery of m, broadcast by *ab(m)*, is visible to a destination process (in other words the implementation of *ab* is irrelevant). The protocol executed by a process P_i is defined as follows [5]:

```
r_i(x)v  = return for v the value of the local copy of x
w_i(x)v  = begin ab(x, v, i); done := false;
                wait(done);
           end

when (x, v, j) is delivered to node i
begin value of the local copy of x := v;
      if j = i then done := true fi;
end
```

It is easy to see that this protocol ensures sequential consistency. All writes on local copies are executed in the same order in all the nodes (due to the *ab* primitive). Furthermore, this protocol implements fast reads (a read is local to a node); duration of writes depends on the implementation of the *ab* primitive.

3.3.4 Protocol based on a central manager

This protocol (a special case of a general protocol described in [11]) assumes an existence of a special node which acts like a central manager for the whole set of objects; the node owns the current copy of each object. To simplify the presentation this node is assumed to be an additional node without being associated with any application process. Let *object_range* be the set of all object names. The central manager is endowed with the following control information:

- an array *current[object_range]*.
  current[x] is the version number of current x.
- a matrix *known_by[1..n, object_range]*.
  known_by[i,x] is the version number of the copy of x, that P_i has in its local memory.

Each write on any object x by a process is sent to a central manager, that updates its copy of x and, by looking up its two tables, checks whether some of the copies of objects that P_i has in its local memory are too old (i.e. inconsistent with respect to the values of current copies); new values are then sent to P_i for such copies.

This protocol can be stated in the following way for a process P_i.

```
r_i(x)v = return for v the value of the local copy of x
w_i(x)v =
begin send (x, v) to the central manager;
      receive (new) from the central manager;
      for all (y, v_y) ∈ new :
            do value of the local copy of y := v_y od;
      value of the local copy of x := v
end
```

When the central manager receives a message (x, v) from a process P_i it executes the following:

```
begin value of the copy of x := v;
      current[x] := current[x] + 1;
      known_by[i, x] := current[x];
      new := φ;
      forall y ∈ object_range :
         do if known_by[i, y] < current[y]
               then new := new ∪ (y, value of y);
                    known_by[i, y] := current[y]
            fi
         od;
      send (new) to P_i
end
```

As does the protocol by Attiya and Welch, this protocol implements sequential consistency with fast read, and totally orders all writes on all objects. In the previous protocol the total order on writes is produced by the atomic broadcast primitive, while in this protocol it is achieved by sequentiality of the central manager that processes one message at a time. Furthermore, this protocol preserves real-time ordering on write operations; if we consider only write operations it implements linearizability.

This protocol can be modified to allow fast write operations: in that case after the update of its local memory, P_i does not wait for a response from the central manager. Instead, it continues its execution. However, when P_i invokes a read operation, it has to wait for responses for all of its previous write operations before executing the read. Finally the protocol can be generalized to the case where local memories include copies of only a dynamically varying subset of the objects (see [11] that proves this general protocol ensures sequential consistency).

4 Hybrid consistency

4.1 About weak consistency criteria

Linearizability (before being precisely formulated and analyzed by Herlihy and Wing [7]) was the classic criterion for objects consistency (it was, and sometimes still is, called atomic consistency). Sequential consistency, introduced informally by Lamport [9], received attention in past years. As mentioned before, these consistency criteria declare any distributed execution σ that produces for each process and each object the same history as the one that could have been produced by a correct sequential execution τ of the same program to be correct. As we have seen some price must be paid to ensure such consistencies: it is the price to build a total order (with an atomic broadcast primitive or a central manager). These consistency criteria are usually referred as *strong consistency*.

In order to obtain more efficient implementations, weaker consistency criteria have been defined: weak ordering, type-specific memory coherence, causal memory, slow memory, release consistency, lazy consistency, processor consistency, weaker memory-access order, etc [1]. All these criteria express some form of *weak consistency*. Some of these definitions are very similar and it can be difficult to see their common and different points. Furthermore, they are generally defined by protocols that implement them, instead of formal descriptions of how read and write operations appear to users.

In order to clarify these criteria, we adopt the point of view and presentation by Attiya and Friedman [4], who provide a general framework, called *hybrid consistency*, offering two types of operations (strong and weak). This framework can not only help to understand existing consistency criteria but also allow a system designer to tailor distributed memory semantics well-suited to his demand. In our point of view, this framework is an Ariadne's clew to escape Minos'labyrinth composed of all these consistency criteria.

Operations (reads and writes on shared objects) are classified as either strong or weak. Hybrid consistency guarantees properties on the order in which these operations appear to be executed at the program level. Then by defining which operations are strong and which are weak, a user can tune the consistency criterion to his own needs.

4.2 Strong and weak operations

Informally hybrid consistency guarantees the following two properties:

- all strong operations appear to be executed in some sequential order.
- if two operations are invoked by the same process and one of them is strong, they appear to be executed in their invocation order.

For each process, all the other processes agree on a total order of strong operations issued by the process, but can disagree on the relative order of its weak operations that appear between two strong operations.

If all operations are weak we obtain the weakest possible consistency. At the other extreme, in which all operations are strong, we obtain strong consistency. In the latter case if we additionally impose the sequential order with respect to real-time order of operations, we obtain linearizability; if the order can differ from real-time order we obtain sequential consistency. The interested reader will find more formal definitions in [4].

4.3 A protocol implementing hybrid consistency

Implementation of strong operations needs global synchronization; on the other hand a weak operation can be executed immediately at the invoking node and disseminated to other nodes without special care. The underlying network has been assumed to be fully connected with FIFO channels (see Section 2.3). This property is used to ensure that messages carrying strong and weak operations issued by a process are received in a correct order at each destination node. A message broadcast by a node is received by all other nodes.

Consider first the implementation of a weak operation p on object x invoked by P_i, i.e. $p_i(x,\ldots)$. This case is easy: the operation is executed locally and broadcast to other nodes that execute it as soon as they receive it (weak read operations need not be broadcast as they do not modify values of objects):

weak operation $p_i(x,\ldots)$ =
  begin $p(x,\ldots)$ *is executed locally*;
    broadcast $(weak, p(x,\ldots))$;
  end
receipt by node i **of** $(weak, p(x,\ldots))$
    execute $p(x,\ldots)$ *on the local copy of* x;

Consider now the implementation of strong operations proposed in [4]; this implementation ensures linearizability for strong operations. The global ordering on these operations is enforced by the use of timestamps. At each node P_i, $expected_i[i]$ stores a logical clock and $expected_i[j]$ is a lower bound on all the logical clock values that node P_i will receive from node P_j. The timestamp of a strong operation issued by

P_i is a pair $(expected_i[i], i)$ and the comparison of two timestamps is defined as follows:

$$(h, i) < (k, j) \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

In order to keep messages that have arrived but cannot yet be interpreted, each node P_i manages a set $pending_i$. To ensure liveness, some *ack* messages carrying clock values are used (as in [8]). Finally a control message (*done*) is sent by a node P_j to a node P_i to inform that the strong operation issued by P_i has been executed by P_j. As soon as P_i knows the strong operation $p(x, \ldots)$ it issued has been executed on all the copies of x, the invocation of $p(x, \ldots)$ is considered to terminate and P_i can proceed (for strong operations only the message *done* from node i is necessary if sequential consistency, instead of linearizability, is to be implemented).

In a more formal way, the implementation of a strong operation $p(x, \ldots)$ invoked by P_i can be described as follows (see [4] for a proof of an improved version of this protocol):

```
strong operation p_i(x,...) =
  begin broadcast(strong, p(x,...), (expected_i[i], i));
        put(p(x,...), (expected_i[i], i)) in pending_i;
        expected_i[i] := expected_i[i] + 1;
        progress;
        wait until n messages done have arrived;
  end

receipt by node i of (strong, p(x,...), (k, j))
  begin put(p(x,...), (k, j)) in pending_i;
     expected_i[j] := k + 1;
     if expected_i[i] < k + 1 then expected_i[i] := k + 1;
                                   broadcast ack(k + 1, i)
     fi;
     progress;
  end

receipt by node i of ack(t, j)
  begin expected_i[j] := t;
     progress;
  end

procedure progress;
  begin continue := true;
    while pending_i ≠ φ and continue
     do let y = (p(x,...), (k, j)) the element of pending_i
                        with the smallest timestamp;
      if k < min_j(expected_i[j])
       then execute locally p(x,...);
        delete y from pending_i;
        send done to node j
       else continue := false
      fi
     enddo
  end
```

4.4 Remarks

In an actual use of the protocol, read operations are generally weak and write operations are strong. It can be seen that the protocol allows the attribute of an operation (strong or weak) to be defined at run-time. In a more general setting, weak and strong attributes are in fact a way to declare which operations are commutative and which are not.

4.5 Integrating causal consistency

Causal consistency was first introduced by Ahamad et al. [2]. It is stronger than weak consistency but weaker than strong consistency. The consistency is based on Lamport's causality rules [8], adapted to read and write operations. Reads have to return values consistent with causally related reads and writes; but writes on an object can be concurrent and perceived differently by two processes. The causality relation, denoted $\xrightarrow{c}$, resulting from an execution is as follows:

- if op_1 and op_2 are invoked by P_i and op_1 occurred first, then $op_1 \xrightarrow{c} op_2$.
- if $op_1 = w_j(x)v$ and $op_2 = r_i(x)v$ then $op_1 \xrightarrow{c} op_2$. (It is supposed all values written are distinct to have a rule independent of a particular application; moreover there is no intervening $w_k(x)u$ such that $op_1 \rightarrow w_k(x)u \rightarrow op_2$ for $\xrightarrow{c}$ to be legal [11]).
- if $op_1 \xrightarrow{c} op_2$ and $op_2 \xrightarrow{c} op_3$ then $op_1 \xrightarrow{c} op_3$.

To illustrate causal consistency consider the following distributed execution σ:

$$\begin{array}{ll} P_i: & w_i(x)a;\ r_i(x)b \\ P_j: & w_j(x)b;\ r_j(x)a \end{array}$$

writes on x are concurrent and P_i and P_j perceive the following sequential executions τ_i and τ_j that are consistent with σ (σ is causally consistent but not sequentially consistent):

$$\begin{array}{ll} \tau_i = & w_i(x)a\ w_j(x)b\ r_i(x)b \\ \tau_j = & w_j(x)b\ w_i(x)a\ r_j(x)a \end{array}$$

In other words, all processes could agree on the same partial order but, as they are sequential, each one has its own sequential perception of this partial order and these sequential perceptions can differ on the order of non causally related operations.

Several protocols have been proposed to implement causal consistency; We consider the one presented in [2]. All nodes receive all writes operations and apply them sequentially in an order consistent with $\xrightarrow{c}$

This partial order $\xrightarrow{c}$ is classically encoded with vector clocks $vc[1..n]$ associated to each write operation; if $vc(op)$ is the timestamp (vector clock) value associated to op we have: $op_1 \rightarrow op_2 \Leftrightarrow vc(op_1) < vc(op_2)$ with

$$vc(op_1) < vc(op_2) \Leftrightarrow \forall k : vc(op_1)[k] \leq vc(op_2)[k] \text{ and } vc(op_1) \neq vc(op_2)$$

The local set $pending_i$ keeps writes received but not yet processed by node i. All local memories contain a copy of each object and a local vector vc_i.

```
r_i(x)v = returns for v the value of the local copy of x
w_i(x)v =
  begin vc_i[i] := vc_i[i] + 1;
        value of the local copy of x := v;
        broadcast(x, v, vc_i, i); progress
  end
receipt by node i of (x, v, vc, j)
  begin vc_i[j] := vc[j];
        put(x, v, vc) in pending_i;
        progress
  end

procedure progress;
 begin continue := true;
  while pending_i ≠ φ and continue
   do let y = (x, v, vc) one of the elements of pending_i
         such that all other elements of pending_i have
         a timestamp vc' with ¬(vc' < vc);
      if vc ≤ vc_i then localcopy of x := v;
                        delete y from pending_i
                   else continue := false
      fi
   enddo
 end
```

As we can see causal consistency is "weak operations" +"causality". Moreover if concurrency on all writes are prevented we get the following relation : "causal consistency"+"total order on all writes on all objects"="strong consistency".

5 Conclusion

This paper surveyed and classified consistency criteria for distributed object memories and showed that hybrid consistency is a good framework to understand all such criteria. Strong consistency is now well understood as it corresponds to the classic common memory multiprocessor run-time model. Some work remains to be done on weak consistency criteria. Among such work, classification of applications based on consistency criteria that they fit in is certainly very important from a theoretical as well as a practical point of view.

References

[1] ADVE S.V., HILL M.D. *Weak ordering: a new definition.* Proc. 17th IEEE Symposium on Comp. Arch., (1990), pp. 2-14.

[2] AHAMAD M., BURNS J.E., HUTTO P.W., NEIGER G. *Causal memory.* Proc. 5th Int. Workshop on Dist. Alg., Delphi Greece, Springer-Verlag LNCS 579, (1991), pp. 9-30.

[3] ARCHIBALD J., BAER J.L. *Cache coherence protocols: evaluation using a multiprocessor simulation model.* ACM Trans. on Comp. Systems, Vol.4,4, (1986), pp. 273-298.

[4] ATTIYA H., FRIEDMAN R. *A correctness condition for high-performance multiprocessors.* Proc. 24th ACM Symposium on Theory of Computing, Victoria CA, (1992), pp. 679-690.

[5] ATTIYA H., WELCH J. *Sequential consistency versus linearizability.* Proc. 3rd ACM Symposium on Par. Algo. and Architectures, (july 1991), pp. 304-315.

[6] BAL H.E., KAASHOEK F., JANSEN J., TANENBAUM A.S. *Replication techniques for speeding up parallel applications on distributed systems.* Concurrency Practice and Experience, Vol.4,5. (1992), pp. 337-355.

[7] HERLIHY M., WING J. *Linearizability : a correctness conditions for concurrent objects.* ACM Trans. on Prog. Languages and Systems, Vol.12,3, (1990), pp. 463-492.

[8] LAMPORT L. *Time, clocks and the ordering of events in a distributed system.* Comm. of the ACM, Vol.21,7, (1978), pp. 558-565.

[9] LAMPORT L. *How to make a multiprocessor computer that correctly executes multiprocess programs.* IEEE Trans. on Computers, Vol.C-28,9. (1979), pp. 690-691.

[10] LI K., HUDAK P. *Memory coherence in shared virtual memory systems.* ACM Trans. on Comp. Systems, Vol.7,4, (1989), pp. 321-359.

[11] MIZUNO M., RAYNAL M., SINGH G., NEILSEN M. *Communication efficient distributed shared memories.* Research Report 691, IRISA, Rennes, (déc. 1992), 21 p.

Unité de Recherche INRIA Rennes
IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399